Parsing the Parser

a case study in programming style

Peter Desain
January 1991

Center for Knowledge Technology
Utrecht School of the Arts
Lange Viestraat 2B
NL-3511 BK Utrecht

City University
Music Department
Northampton Square
UK-London EC1V OHB

Abstract
The value of the computational approach in the cognitive sciences lays both in the need to formalize theories such that they can be implemented as computer programs and in the subsequent ease of experimenting with these programs. In this paper I hope to show that, the cleaner a programming style is used, the more these benefits will be present. As an example, the musical parser designed and described by Longuet-Higgins (Longuet-Higgins 1976, 1979) is re-implemented in a clean functional programming style in LISP. This yields a, so called, micro version that makes the theoretical issues that the original program was supposed to illustrate, stand out much more clearly.

Introduction
Much of my effort during the last years was directed at explaining neat programming styles and functional use of programming languages (Desain 1990). My audience however tended to question the realism of the beautiful three line programs I mostly used as examples. They doubt whether they were not just of a didactical worth, real world problems being to hard to handle without compromise in this way. To contradict such insinuations I went looking for a really complex problem, in the form of a published program which I could convert into a good functional programming style. In research in expressive timing in music I came across the excellent papers of Longuet-Higgins (Longuet-Higgins 1976, 1979) describing a musical parser that, next to tonal analysis, parsed performed music rhythmically. It produced a metrical hierarchical structure, while tracking tempo changes and rounding performance inaccuracies. The actual code of the program written in POP-2 was attached to the article as an appendix, which made the proposed endeavor possible.

Researchers should really be encouraged to publish the actual code or parts thereof, like Longuet-Higgins did. Firstly it provides a means to verify or falsify the claimed results. Secondly it forces the author to account for every detail in the system. Especially if the algorithm is claimed to provide a cognitive model, it is important to study its internals, the data and control structures used, so as to be able to state the predictions it makes. Naturally, bulky programs are not useful as appendices to articles as usually nobody takes the trouble to look at them. Of more use are micro versions, from which unnecessary details are removed. Constructing such a micro version can be of benefit to the researcher too, being forced to decide what is essential and what are mere 'bells and whistles'. More than once I witnessed remarkable progress in research caused by the insight yielded while trimming a program to its bare minimum. In [Shank and Riesbeck, 1981] good examples can be found of micro versions of some famous computer understanding programs.

In the case of the Longuet-Higgins parser it seemed that the code could indeed already be called a micro version, apart from the fact that the tonal and the rhythmical analysis, which are being dealt with separately in the theory, were embodied in one program. However, speaking to several colleagues that had tried to understand the code, I discovered that the program did not at all function as a clarifying, and helpful addendum to the article itself. All readers (including myself) were put off by the difficulty of the program, even though the underlying theory was described extremely well. This was not because it was written in the (now obsolete) language POP-2, but because the program itself used many awkward programming constructs: side effects, different binding regimes and scoping rules, non-local exits and even a GOTO. The term spaghetti program seems to be a good description of this piece of code (and indeed prof. Longuet-Higgins is fond of Italian food), and one has to have an appetite for reverse engineering to rediscover the workings of the program from the code. But at least there was well described code available, which cannot be said of all publications about AI programs. It is a symptom of the general state of AI research that rational reconstruction is becoming an significant AI methodology. Campbell (1990) defines this technique as reproducing the essence of a programs behavior with another program constructed from descriptions of the purportedly important aspects of the original, trying to verify claims made about this program. It seems a waste of effort as these programs should have been published and described well in the first place, but it makes again clear that the equation 'the program is the theory' that has had a long standing history in AI, does not hold.

When finally Edward Lisle, Longuet-Higgins present collaborator, remarked that I would never be able to port the code to LISP because one needs to be an expert LISP programmer to do that, I had gathered enough incentive to embark upon the task of rewriting the program in an understandable style. In this paper I will describe the route I took in porting the program, in the hope that similar methods will be useful for the reader on other occasions. I will also show how the standard repertoire of the LISP and AI programmer can be used to create elegant and modular programs for complex problems. The resulting code is rather easy to read for humans - which is the main, but often forgotten, aim of programming - and can be much more easily experimented with, changed and tested.

But before I embark upon describing the program and the port, I first have to make clear that looking at the model so closely only boosted my admiration for the research itself. And although the flow of data and control needed for the theory is rather sophisticated, the questions involved are described very well in the two papers, and the performance of the algorithm is remarkable. Furthermore the code was written almost two decades ago when lots of the techniques I used, now common practice, had not yet found their way into the literature. Prof. Longuet-Higgins was so kind to encourage me on this task and clarify several issues. Any criticism in this article, in which I cannot hope to match his personal eloquent style and merciless polemics (see Longuet-Higgins, 1983), has to be seen in the light of these remarks.

Understanding the theory

The parsing and quantization process is known to be very hard. Different methods can be found in (Desain & Honing 1989, 1991) and a comparison of the performance of Longuet-Higgins' symbolic method and a connectionist model for the same task is given in (Desain, 1991). But first and foremost the reader must be asked to read the original papers, or the corresponding chapters of (Longuet-Higgins 1987), as I can only give a brief outline of the theory here.

The rhythmical part of the parser uses a hybrid method of tempo tracking plus the use of structural knowledge about meter. In this method the tempo tracking is done with respect to a time interval that can span zero or more notes. At top level this time interval represents a beat. It is subdivided recursively in 2 or 3 parts looking for onset times near the start of each part, until the interval contains no more onsets. The 'best' subdivision is returned, but the program is 'reluctant' to change the number of subdivisions (the pulse) at each level. At each recursive level the interval length is adjusted on the basis of the onsets found, just as in simple tempo tracking methods. An articulation analysis is then performed, dividing notes into tied parts and deciding where a rest occurs. Next to the quantized results this program delivers a hierarchical metrical analysis, whose top level is the beat and whose bottom level are made up of notes and rests. From the article we can identify the input and output of the system, the data-types used, the parameters, the procedural modules and their communication. What follows is an outline of those issues, taking into account only the relevant ones from a rhythmical perspective.

The input of the system consists of an ordered list of notes. Each note has an onset time, an offset time and a pitch. The output of the system consists of a list of trees, one for every analyzed beat. A beat is just a period of time, slicing the data in consecutive intervals. Each tree is of a combined binary-ternairy nature, which means that each node has zero (in case it is a leaf of the tree) or two or three sub-trees. The arity of each internal node is called the pulse. During the construction of the tree there is a horizontal flow of pulse information through the layers of the tree, seeking to maintain the same pulse at a certain level as long as possible. The list of proposed pulses for the tree at each level is called meter. During the construction of the tree a strict left to right order is maintained, and new sub-trees are created on a generate and test basis. This means that a proposed (and constructed) binary sub-tree may be rejected in favour of a tertiary one. The generate and test procedure is non-standard in that it may, after checking and rejecting the first alternative, still reject the second in which case as yet the first alternative is chosen. Notes, whose onsets happen in rapid succession (like in a trill) are collected in a group and treated as if they started at the onset of the first note in the group. Associated with each leaf of the resulting tree is a possibly empty, annotated list of sounding notes. There is one parameter identified in the program called tolerance which is used in different places as the allowed margin of deviation in deciding if notes start or stop at a certain times.

The main flow of control is dealt with in a mutual recursion of the procedures **tempo** and **rhythm**. **Tempo** decides if there is another subdivision needed, if not it calls **singlet** (bottom case of recursion). If there is, it calls **rhythm** which tries one or more subdivisions and calls **tempo** recursively on them. **Rhythm** then returns the best fitting sub-tree. There is some pre-processing (**sift**, **takein**), some top level initialization, (**startup**) and the post-processing is mainly done in the form of printing procedures (**typeout**, **reveal** and **describe**). The call-graph (it is not a hierarchy because of the recursion) depicts it all neatly (Figure 1).

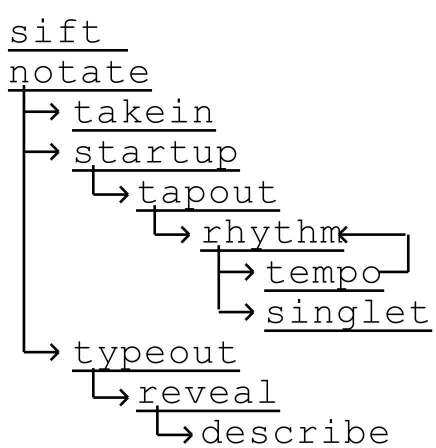

Figure 1. *Call-graph of the parser routines.*

Planning the endeavor

Because the flow of data and control in the program is so complicated, and because a complete rewrite from scratch could only use the text of the articles as specification, which seemed not enough, I planned the whole endeavor as follows. I would try first to reconstruct the system as a LISP program directly translating the POP-2 constructs into LISP and staying as close as possible to the original code. During that stage some of the example input data incorporated in the articles could be used to test the program. It was planned to do only one change at a time and to keep all intermediate files for easy recovery and documentation. Every question about the working of the program was added as comment to those files. During that stage I would try to resist improving the code making the translation as algorithmic as possible. It was decided to concentrate on the rhythmic part, leaving the simpler harmonic analysis for the future. When that program would work well a test suite had to be built to check the output using all of the available examples. Those two means, trying to translate the POP-2 code as directly (and mechanically) as possible and checking the input/output of this program, would hopefully insure a version (called LISP Program 1) that was semantically equivalent to the original. However, "testing can show the presence of bugs but not their absence" (Dijkstra in Bentley 1988, p. 60). I planned then to add some trace code to check the internal workings of the program and clarify the list of questions that was building up. The tracing code and the test suite all belong to the necessary scaffolding that has to be erected when building or modifying a program (Bentley 1988). Although these techniques are common practice for every experienced programmer it is a pity that they are so often neglected in student texts on programming, and that support facilities for these temporary constructs are lacking from almost all programming environments. In the next stage some non-essential add-ons could be removed and then enormous changes would be necessary to clean up the code. Only semantic invariant program transformations were to be used, insuring that, however different its appearance, the behavior of the program was not changed by the surgery: it would still exhibit the same input-output behavior. After each change that would be checked, using a test run of the program suite. After resulting in a clean functional program (LISP Program 2) I suspected that the internal flow of information and control would be so much clarified that the remaining questions about the internal workings could be answered and the crucial theoretical concepts could be made apparent from the code itself. Then at last one might be able to point at possible improvements of the algorithm. With this plan in mind the next stage was started.

Literal transportation.

When starting this project I was not able to locate a POP-2 manual. Afraid that the whole project would turn out to be a piece of computer science archeology I was glad to find that POP-11 (the successor of POP-2) is still widely used and a manual (Barett e.a. 1985) only left a few constructs found in the program unexplained. When I eventually found the POP-2 manual it was quit instructive to see the sloppy semantics (Burstall e.a. 1986 reference manual p. 14-15) of this language, which was used for a lot of large programming projects.and has had a great influence in the AI-community in Britain for a long time. Common LISP (Steele, 1984) was chosen as the LISP dialect because of the wide availability of implementations of this standard, although SCHEME (Abelson & Sussman, 1985) would yield even more elegant code.

We will now take a dive into the details of the POP-2 and LISP code, anyone interested in a more global view may skip this part and start reading again at the section 'Lispizing the code' or even move ahead to 'Theoretical issues'.

The relevant parts of the POP-2 code are shown in appendix 1. I have inserted some comments (printed in italics) and added the line numbering. Any line numbers in the text refer to this appendix. Translations of POP-2 constructs in LISP are shown in Table 1.

construct	POP-2 syntax	line	LISP translation
assignment	`<expression> -> <variable>;`	52	`(setf <variable><expression>)`

conditional	`if <condition>`
	`  then <statement1>`
	`  else <statement2>`
	`close;`
		53	`(if <condition>`
			`    <statement1>`
			`    <statement2>)`

multiple conditional	`if <condition1>`
	`  then <statement1>`
	`  elseif <condition2>`
	`  then <statement2>`
	`  else <statement3>`
	`close;`
		125	`(cond (<condition1>`
			`<statement1>)`
			`      (<condition2>`
			`<statement2>)`
			`      (t  <statement3>))`

iteration	`loopif <condition>`
	`  then <statement>`
	`close;`
		25	`(loop`
			`  (when <condition>(return))`
			`  <statement>)`

goto	`...`
	`<label>:`
	`...`
	`goto <label>`
	`...`
		75,
		99
			`(prog ()`
			`  ...`
			`  <label>`
			`  ...`
			`  (goto <label>)`
			`  ...)`

function application	`<function>(<argument1>,`
	`           ...,`
	`           <argumentn>);`
	or
	`<argument>.<function>;`
	or
	`.<function>;`
		85
		
		135
		
		151
			`(<function> <argument1>`
			`            ...`
			`            <argumentn>)`
			
			`(<function> <argument>)`
			
			`(<function>)`

function definition	`function <function>`
	`  <parameter list>`
	`  -> <output local list>`
	`  vars <local list>;`
	`  <body>`
	`end;`
	
	`=>` *can be used instead of* `->`
		51
		
		2
			see Table 2

constant list	`[...]`	125	`'(...)`
list selector	`hd`	4	`first` *or* `car`
list selector	`tl`	4	`rest` *or* `cdr`
list constructor	`::`	40	`cons`
list constructor	`[%...]`	42	`(list ...)`
list mapping	`maplist`	129	`mapcar`
list iteration	`applist`	128	
list reversal	`rev`	117	`reverse`
record type declaration	`recordclass class slot1 ... slotn`	1	`(defstruct class slot1 ... slotn)`
function composition	`<>`	127	
output	`pr`	145	`print`
output new line	`nl`	141	`terpri`
pushing the stack	`<expression>`		

popping the stack	`->`
	or
	`.<function>`
		
		145	

temporary use of stack	`if <condition>`
	`then expression1`
	`else expression2`
		112	`(setf var (if <condition>`
			`              expression1`
			`              expression2))`

	close -> var; <list>.destlist -><variable1> ... -><variablen> <function call> -> <variable1> ... -><variablen>	 101 85	*or* (if <condition> (setf var expression1) (setf var expression2)) (setf <variable1> (first <list>) .... <variablen> (<nth> <list>)) (multiple-value-setq(<variable 1> ... <variablen>) <function call>)
variable declaration	vars <variable1> ...<variablen>;	17	(defvar <variable1>) ... (defvar <variablen>)
identity function	identfn	10	identity

Table 1. *Translation of relevant POP-2 constructs into LISP.*

Points that may need further clarification are the following. Statements in POP-2 are separated by semi-colons. In list processing POP-2 only has syntactic differences from LISP. Function application (a function call) has a prefix syntax with arguments in brackets or a postfix syntax with a dot separating argument and function. In POP-2 values can be left on an implicit global stack by any expression which yields a value. They can be popped off the stack and assigned directly to a variable or used as the arguments of a function. Happily in the program there were no values put on, and popped off the stack in a dynamic way (determined by program flow of control). For an example of typical short time static use of the stack in a conditional, a destructuring bind (assigning subsequent values from a list to several variables) and the return of multiple results by a function see Table 1 and the corresponding lines of code. The return of multiple results is done by the declaration of so called output locals in the function definition. These act as local variables, but on returning from a function call their values are left on the stack. These are best removed from the stack and assigned to variables immediately after the function call, as is done indeed in the program. Only in the lines 65 to 68 the results of **singlet** or **tempo** are left on the stack slightly longer. The return of multiple results by a function is in LISP supported by the **values** construct. They can be caught by the caller using the **multiple-value-setq** assignment. Multiple values are not a really orthogonal designed construct in LISP, but they are saver then the POP-2 solution, as multiple values cannot be put on or popped of the stack at random times. Using this translation one has to be careful with non local exits as in line 95 where POP-2 implicitly leaves the current values of the output locals on the stack, where in LISP one has to return them explicitly.

The handling of variables (the binding regime) in POP-2 is completely clumsy and idiosyncratic. A function can have arguments, output locals and local variables. In line 105 **nlist** is a formal argument of the function **tapout**, **sequence** is its output local, and there is a list of local variables **start**, **beat** etc. The strange thing is that locals are given a value upon entry of the function. As can be seen in line 35 where **stop** and **period** are referred to before ever having received a value in the body of **singlet**. The values they are initialized to are the values of the corresponding variables in the calling environment: the output locals of **rhythm** declared in line 51. This ugly construct makes it impossible to study the behavior of a function in isolation - one has to search always for the part of the program that happens to call the function to decide upon the values of these variables on entry. In (Barett e.a 1985, p 37,38) this is called a convenient feature. Which only once again shows that one has to take care for the words like "handy" or "convenient". They inevitably signal danger when they occur in the description of the semantics of a programming language. A related problem is the fact that assignment to a local variable does not change the value of the variable with the same name in the calling context, but an assignment to a free variable (a variable that is not declared as argument or as output local or local), does change the value of the corresponding variable in the calling code. So only after scanning the whole program one can decide that the assignment to **nlist** in line 54 of **rhythm** is really an assignment to the the **nlist** argument of **tapout** in line 105 just because **rhythm** happens to be called in **tapout**. Such so called dynamic scoping makes the behavior of a function depend upon the actual coding of the functions it uses and by which it is used, complicating the semantics of the language, and of any program written in it.

Things become worse when programmers do not understand these constructs or use them in a sloppy way: why is there a **tol** local variable in line 106, while there is also a global variable **tol** in line 30 and it is used as a truly global constant? Why is there a local **stop** in line 51 when it is also declared as output local in the same line? Why is there a global **metre**, even declared twice in line 30 and 17, when it is clearly used in local backtracking in line 101 and 97? Indeed the whole program is a sloppy mess regarding scope and binding of variables. The translation of these aspects was the most difficult part of the port and I will try to outline how I tried to make all communication between parts of the program explicit and lexical, which means that only references and assignments are made to variables that are local (and textually visible) to the piece of code under construction. All the semantics then become static, which means that the meaning of a part of the program can be described independently from the actual computational route taken by calling and called routines, only depending upon the program-text of those routines. Let us consider the example function definition in Table 2 in which every possible use of variables is listed.

construct	POP-2	LISP translation

function definition	`function fun a b -> c d`
	` vars e f;`
	`  body with`
	`    g referred to but not assigned to`
	`    h referred to and assigned to`
	`    c,e referred to before assigned to`
	`    d,f assigned to before referred to`
	`end;`
		`(defun fun (a b c e g h)`
		`  (let (d f)`
		`     translation of body`
		`    (values c d h)))`

function call	`fun(i,j)->k->l;`	`(multiple-value-setq (k l h)` `   (fun i j c e g h))`

Table 2. *Translation of a POP-2 function into LISP.*

In the function body there are formal arguments, output locals, locals, and free variables They are used (referred to and assigned to) in different order. In the translation into LISP we have to add formal arguments to the function for some output locals and free variables, because they will be assigned an initial value by POP-2 upon entry of the function. Now we will do that initialization explicitly by adding them to the argument list in the function call. The output locals will have their values returned as the multiple results, which has to be done explicitly in LISP. Since the assignment to a free variable in the body will have effect in the calling context as well, we have to add this variable to the multiple results as well and do the assignment explicitly in the calling program. Note that not in each routine of the POP-2 code all the ways of treating variables are used, yielding a simpler translation. But as in general we now have added an assignment in the calling context, the caller may again change its translation, initiating more changes etc. Consequently the translation is not a very simple task. But after this translation all flow of information is clear and we can get rid of some the remaining global variables because the routines will be explicitly passed their values as arguments when they need them. Only **metre** and **tol** remain global and are declared once at the begin of the program text to retain for the moment the spirit of their initialization (line 30).

For individual note a record datatype was used, which can be declared in POP-2 with the **recordclass** construct (line 1) which automatically defines accessor functions for each field of the record (in our case **onset**, **pitch** and **offset** in lines 21, 22 and 35) and a constructor function (in our case **consnote**, line 10). I took the liberty of defining lists of note structures directly (see the end of Appendix 3), without reading an input file, and passing one such list as argument to **startup** thereby removing the need for the **takein** procedure (line 150). Pitch names were inserted in the data, because it is then easy to check the output against the output shown in the articles, even though I left out the tonal analysis. The examples given in the articles are a simple musical cliché and two fragments of the cor anglais solo in the Prelude to Act III of Wagner's *Tristan und Isolde*. All data is given in Appendix 2 retaining the original notation for pitches.

I could not resist the temptation to re-order the procedure definitions top down (using the call graph) and to separate them in several groups. Although this obscured the relation of the program with its POP-2 parent it is so much easier to navigate through code that is ordered well. The resulting program (LISP program 1) produced the same results as shown in the original article and I can assure you that I felt great relieve the moment I saw it parsing the input correctly. The only difference with the output listed in the articles is the output of the first 'count-down' beats, not shown in the original article. Later Christopher Longuet-Higgins affirmed me that his program does output these as spurious rests at the beginning of its parse. Finally I have to mention one typographical error in the original program: the comma in line 21 should be a period.

Lispizing the code

Now the program could be trimmed to remove all aspects that were not to be part of a real micro version. For example the function **sift** is a trivial function to remove spurious key bounces that stem from the recording equipment used. The article should mention such pre-processing but it surely is no part of the parse algorithm itself, and it has even less relevance for the cognitive model. Another feature that should not be in the micro version is the grouping of a number of beats on one output line, faking an analysis above the beat level, while there is just a clever trick used: the musical data is played preceded by a measure of count down beats on the same low key, the number of which is used to collect the beats in measures on the output after analysis. This trick, explained in the article, could be well worth its value in a practical implementation, but it is again far from central to the theory and only distracts the reader of the program. The length of the last count down beat is used as a initial estimate for the beat length. This parameter of the program is thus concealed in the data, it will be cumbersome to experiment with different slices of data, or different initial estimates of the beat. But what is worse, a real issue that the theory does not tackle: how do human listeners pick up the initial beat of a piece of music, is hidden from view by inserting this information in the musical data. It must be said that it may be a wise decision to leave this difficult question aside in the theory, and the article is quite explicit about that, but then again it should be as easy to understand that fact from the program itself. Thus initial beat duration is changed into a parameter of the top level function (and for compatibility with the old data, it is optional and uses the old method if not specified).

As a general rule it is best not to do much processing in routines that produce text, because in textual output all internal structure is lost and other programs often cannot make use of the results in flat text format. In our case the scaffolding, like the test suite and programs that measure the sensitivity of the parser to parameter changes, is much easier to write if they can inspect the whole result structure from the parser. So any output side effects like printing results in a neat way should be postponed. And, if they are included at all, they should be written as an almost trivial add-on. Because the built-in LISP pretty printer (**pprint**) can do a nice textual layout of the result of the parser, handling indentation etc, I decided that in this case the parser should behave as a real function without side effects (note list in, structure out and no printing going on), and I moved the post-processing inside the parser itself.

As decided, all further transformations were done as semantic invariant program transformations. Examples of such transformations that retain the behavior of a piece of code but change its form, are the substitution of a function call for its body (with appropriate substitutions of variables), the collection of statements into a (help)function, the 'unwinding' of loops, the movement of statements in or out conditionals and function bodies, the removal of uneffective assignments, the change of order of independent statements, and the systematic change of variable names. To begin with the latter: some abbreviations of variable names seem silly (**syncop** instead of **syncope, tollerance** instead of **tol**, etc.), I changed these all to there full names, but I did not consider changing them to names that described their role better, nor did I change the name of any routines, so as to maintain the relation with the original program.

Since **metre** was already an argument to the main parsing routines - stemming from the translation of function definitions, I could remove it completely as global variable, and turn it into a parameter of the top level **notate** function. And indeed, just like the initially expected beat period, the expected meter is conceptually an argument of the parser. The same was done for the **tollerance** parameter. This clean-up of global variables made the whole **startup** routine superfluous.

Rewriting **tapout** as a recursive procedure, would made the output local **sequence** as a temporary hold of the growing structure unnecessary, and also would automatically built the structure in the correct order such that a final reverse (line 117) becomes obsolete. These simplifications come often for free when turning iteration into recursion. Using the loop macro, as was done here has the same benefits, and made the tapout function superfluous.

Some routines should really have been carved up into smaller units, either because they are just to large to comprehend as a whole, or because really a separate theoretical issue was being dealt with and modularization would show more clearly on which information the decisions taken were based. E.g. in lines 34 to 45 an analysis of the type of articulation is taking place intertwined with some maintenance of data-structures (e.g. **last**: list of pending notes, and **group**). A simple separation of concern as implemented in the extra help function **articulation-mark**, makes it completely clear that the decision on the type of articulation is taken not on the basis of the gaps between notes, but on the basis of the gap between the end of a note and the beginning of the next metrical unit (Longuet-Higgins, 1987 p. 127).

The collection of notes from **note-list** in a group in lines 53 to 64 in **rhythm** is another complicated piece of code which deserves to be separated and cleaned up (the resulting function is called **collect-group**). Of course the **goto** construction (line 55) is obscuring and completely unnecessary. If you are not convinced of this: (Dijkstra, 1968) is the standard text to explain the horrors of **goto**'s.

Defining helping functions (**onset-before** and **offset-before**) for deciding if the onset or offset of a note occurs before a certain point in time with a margin of tollerance in a certain direction, a frequent operation in the code, again makes the program more readable.

We can keep the data representation of a group a bit closer to the problem at hand. For efficiency reasons a group of notes is represented backwards in the original code. This obscures the calculation of the articulation mark (the actual mark is calculated on the basis of the latest note in a group, not on the basis of the first one as line 35 seems to suggest. Furthermore, in a later stage and in an unrelated piece of program text, this reverse coding has to be undone (line 40). In micro programs one should not worry about tiny gains in processing speed but either keep the data structures as close as possible to a 'natural' representation of the problem at hand - or hide an encoding in a data abstraction layer.

One further problem shows when looking closely at the code. In lines 57 to 60 subsequent notes are removed from the **note-list** and put into the **group**. But in line 62 an actual undoing of the last of such actions might happen. This has severe consequences for high level descriptions of the parser as a process in which a stream of notes is fed in and a parsed structure per beat comes out. Unreading a stream is a rather awkward operation and might be psychologically unplausible. However, given the current algorithm I couldn't do better then change a 'read' plus subsequent 'unread' into one 'peek' operation. Having a more decent sense of a timed input stream, it might be possible to judge on the basis of the offset of the previous note and the current time if no note-onset had arrived yet and the group may be closed and processed.

We have now arrived at the most difficult part of the code: **tempo**. Firstly one can see that although **again** is given a numerical value that is incremented each time through the loop, the loop is done twice at most. So **again** can be changed to a boolean. It is a common error in programming to use under-restricted data types. But to get some grip on the code it is better to unwind the loop, writing down its body twice and get rid of the **again** variable and the non-local exit of line 95. Looking through the processing of **pulse** and **metre**, which is done in an incredibly ugly way in lines 76, 79, 94, 97, 99, and 101, and the clumsy storing and retrieving of local state in lines 74 and 98, the control structure slowly emerges. A trial is done with initial values, if it succeeds, its results are returned. If it does not, its results are stored in a variable called **new** and a second trial is initiated with initial values reset to their old values, but with an alternative **metre**. If this one succeeds, its results are returned, otherwise the results of the first trial is preferred because it used the expected meter. This generate-and-test process calls for a help function (named **trial**) which might be called twice with partly the same arguments, to generate the alternatives, elevating the need for resetting variables to their old values. Now **old** can be removed. Making the control structure stand out clearly in this way facilitates discussions about its nature and the cognitive plausibility of such constructs. It also enables the design of custom flow of control, which can be done in LISP by adding continuations to the **trial** function - which are functional arguments that specify what should be done with the result of it - (Abelson & Sussman. 1985), with a new control structure programmed as a function with functional arguments to specify the details, or with the general macro facilities. The latter actually specifies an extra layer, a new programming language, in which the program is embedded. I choose here for the second construct because it completely isolates the control issue but is somewhat easier to write than macros. Note that the control structure cannot be called proper backtracking because failure or success is decided at the top level of each decision-subtree. It is implemented in the new function **generate-and-test**, which is given a way to generate a result (the function **trial**), a way to evaluate this result (made by **make-test**), a way to make alternative arguments for the generator (called **alternative**), and some initial arguments for the generator to use in the first trial. It will return either the result of the first or of the second trial according to the rules given above.

Some details about the body of **tempo** still have to be clarified. In line 99 there is a expression (**5 - pulse**) which completely obscures the fact that pulse in the program can only be **2** or **3**, and the effect of this expression is the switch from one to the other. Conceptually 2 and 3 as possible values of pulse are not related by their sum being 5 but by being the two smallest primes. Cleverness like this should be abolished from all micro programs (maybe even from all programs). By modularizing this operation into a function **alternative-metre** that calculates a changed meter (not just a changed pulse) a theoretical issue is again highlighted: in changing meter at a certain level, one disposes of all metrical structure below that level - which will again default to divisions into two, a assertion that clearly has cognitive relevance and can be tested as such. Modularizing the test for acceptability of a result into one function (made by **make-test**) again makes part of the theory stand out, showing that a different meter is tried if the metrical unit fails to end with a note (a syncope), or ends rather early or late (Longuet-Higgins, 1987, p. 129). The method of tempo tracking used in **tempo** stands out more clearly now. An extra parameter **speed** will make one more hidden parameter explicit: the speed of tracking tempo at beat level, taken as a constant 2 in line 113). The speed at lower levels can be seen to be equal to 1/pulse.

Because in the program most assignments are now assured to have only local effects within function bodies and are done at most once, I could gradually change them into local binding constructs which made this even more clear. I changed **multiple-value-setq**'s into **multiple-value-bind**'s, moved initial assignments to local variables into the **let** headings where they were declared, and ended up with a program which was side-effect free except for two multiple-value-setq's within a loop construct. This means that if spotting a variable referred to in the program, I could be sure that it was given an initial value only once and see immediately from the locally surrounding program text how that was done, and anywhere within the scope of that construct this variable would retain this value. Thus a computational variable would look much more like a mathematical one, and the actual dynamic aspects of computation-steps taken are now separated from- and irrelevant for- these issues.

The value of **stop**, returned from **rhythm** can be 0 (line 87), in which case it is used as a flag to indicate a detected syncope. It is in general unwise to store conceptually different types of information in one variable. So I made an extra result variable called syncope. Now one can return also a useful stop result in case of syncope: the initially estimated end of the unit. Moving even the tempo-track calculation to a lower level made some more code simpler, to the expense of having to pass the estimated end of the unit (**aim**) and the tempo-track speed (**speed**) downward.

The next major surgery was the un-merging of structural analysis (based on the onset of notes) and articulation analysis (based on their offsets). Although they both use information about the tree being constructed (like **start**, **stop** and **period**) that is only available temporarily (during the local construction), I did feel that the maintenance of the list of still sounding notes in the **last** variable obscured the working of the structural analysis. And because it is well know that merging different algorithms into one is one of the main sources of bugs and confusion in programming, I decided to un-merge the algorithms. This would have the added advantage of making them available as separate modules. **Tapout** will now deliver tree structures in which the leaves contain only note groups whose onset start there but which are annotated with the extra information needed by the articulation analyzer. And because part of the articulation analysis was already done at a later stage (during the printing in **describe**), moving all of it to a separate module did not seem such a essential change. There are of course cognitive arguments to consider the two processes as intertwined, but then again one could consider the program as being implemented on a lazy evaluator which would only do a round of structural analysis only when the articulation analysis needed the result, thus eliminating any psychologically unplausible long-term intermediate storage. It is very difficult to describe the relation of a program to a cognitive model, especially to describe where and how the algorithm and the language semantics over-restrict the model (describe the model in more detail then intended) and I strongly disagree with any indication of ducking these issues as in:

> *The program is, of course, no more than an embodiment of these ideas in computational form*
> (Longuet-Higgins 1987, page 183)

Although it has to be said that at the time of the original paper the very idea that a program can be a medium for expressing ideas about cognition was a novel one.

The last changes delivered the final code: LISP program 2 (see Appendix 2 for the full code plus an example of its use).

One of the most heard (and silliest) arguments used against a clean programming style is the supposed expense in calculation speed and memory usage. This argument was again proven false by this program that performs even faster than version 1 (using the three examples, without output printing, running on a Mac IIci in Allegro Common Lisp).

The test suite I used during the transformations described above was quite small. It consisted of the three examples shown in the articles and given in Appendix 3. Because I suspected that subtle bugs e.g. in the tempo tracking might produce correct results on these examples but on the basis of wrongly calculated internal values, I added cases in which the tolerance was just so far off that the program came up with a wrong answer. The test suite then used that value plus the wrong answer as a reference to judge a correct working of the modified program. This way I could catch any subtle errors in my port which would otherwise go unnoticed because of the rounding mechanism. For one aspect of the program representative examples were missing: **collect-group** only once comes up with a group of two notes (in example tris: the notes at 1928 and 1932 centiseconds). The examples contain just not enough trills, grace notes etc. to test its working thoroughly.

Because the number of arguments to functions is large and the data structures passed around may also be large, build-in trace facilities bury one in pages of text. I designed a custom tracer that produced only the relevant information. Because this is still a lot of information, a graphical trace program would be desirable

Theoretical issues
parameters
The different settings for the tolerance parameter used in parsing the examples in the article (0.10 sec. for the cliche example and 0.13 sec. for the others) raise the question how sensitive the model is for its parameters. It is easy to do an experiment to check the range of parameter values in which the parser works well for the examples. In Figure 2 these ranges are shown for the tolerance parameter, with the initial beat estimate used as a second independent variable because it may disturb correct parsing.

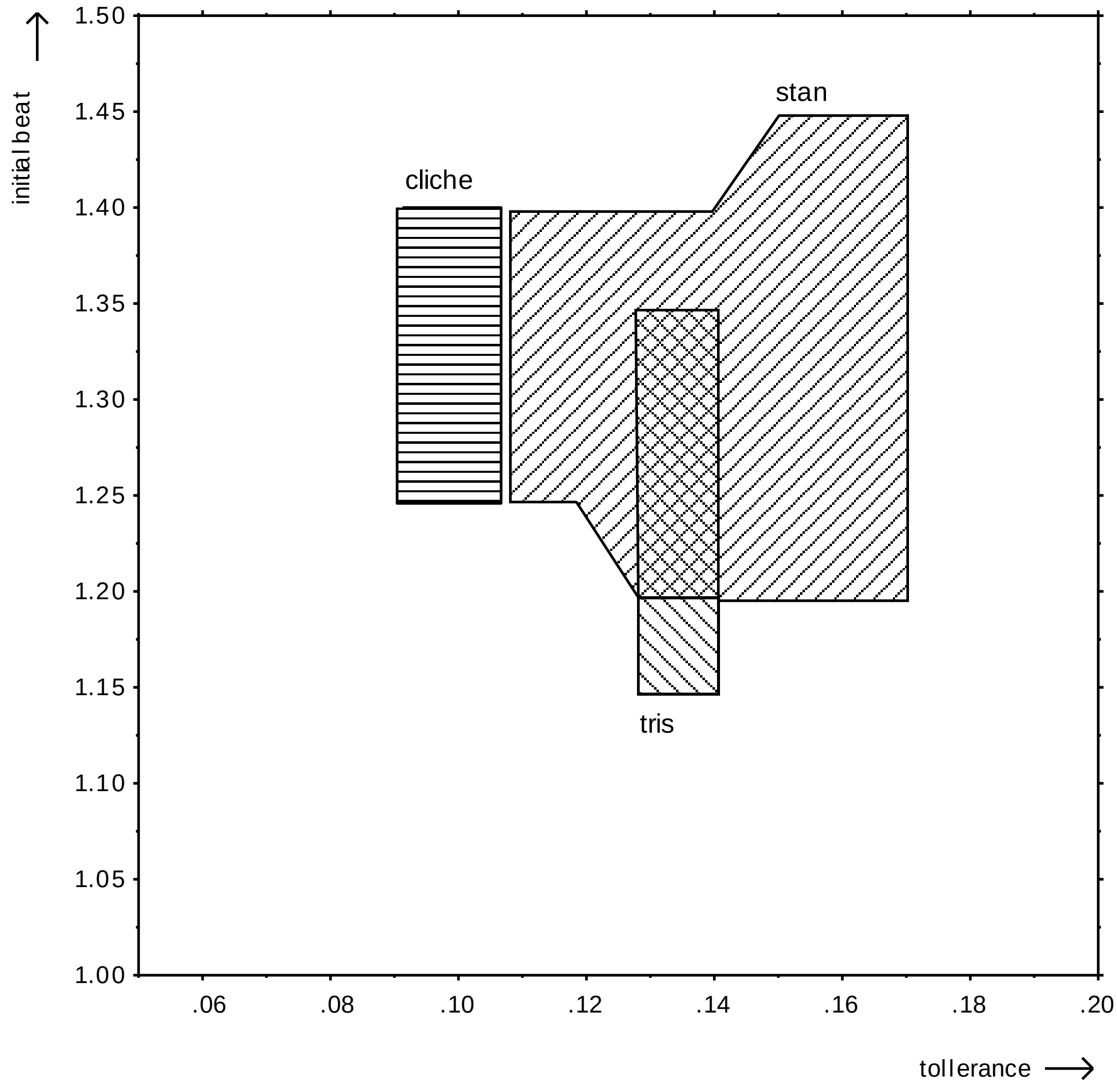

Figure 2. *Parameter ranges resulting in a correct parse.*

It is hard to base a conclusion on the basis of this limited set of three musical examples, but the small size and the non-overlapping nature of the regions identify a problem concerning robustness here. A delicate parameter setting, to be done anew for each piece of data, may be justifiable in the context of a technical tool, it is not so in the context of a cognitive model. These maps show how the initial beat estimate is more or less independent of these results. Thus the tempo tracking taking place at the highest (beat) level is not the source of the problems, but the processing at deeper levels of subdivision is. The same conclusion can be drawn from the fact that allowed settings for the speed parameter in the succesful regions are almost unrestricted within its full range between zero and one (not shown). This may indicate that for the data given, their is no heavy reliance on beat level tempo tracking. For definitive evaluations more data has to be used, systematically mapping out the parameter space of the algorithm (parameter setting versus percentage parsed correctly). A line of further research that may make the parser more robust is the adaptation of the tollerance at deeper levels.

<u>tempo tracking</u>

The tempo tracking at the highest (beat) level is implemented in lines 113. It simply averages the expected beat length and the measured one if the latter is available. Around line 88 a complex process controls the tempo tracking at deeper levels of subdivision. It incrementally adds each deviation found in the subdivision to the total period, and proportionally divides this period to estimate the position of the onset ending the next sub-period. Onsets are allowed within the tollerance around each estimate. For a three-division this effectively amounts to a positive feedback from the timing of second to the third onset. E.g. if the onset ending the first subdivision is too early, the next onset is expected early as well (by 2/3 of error of the first onset). This process goes on, and assuming the second is early as well, the third is expected to be even earlier. This would yield nonsensical behavior were it not that after the completion of the parse of each subdivision the total length is passed one level upwards and compared with the beat estimate at that level, allowing for a deviation of tollerance. So here it turns out that after two short sub-divions the third should be long to pass this test. Because in the parser the tempo tracking mechanism interacts with the change of meter decisions it is quite hard to derive at the mathematical characterization of the set of performed temporal patterns that will be recognized by the parser as a triplet, but testing this empirically is quite simple.

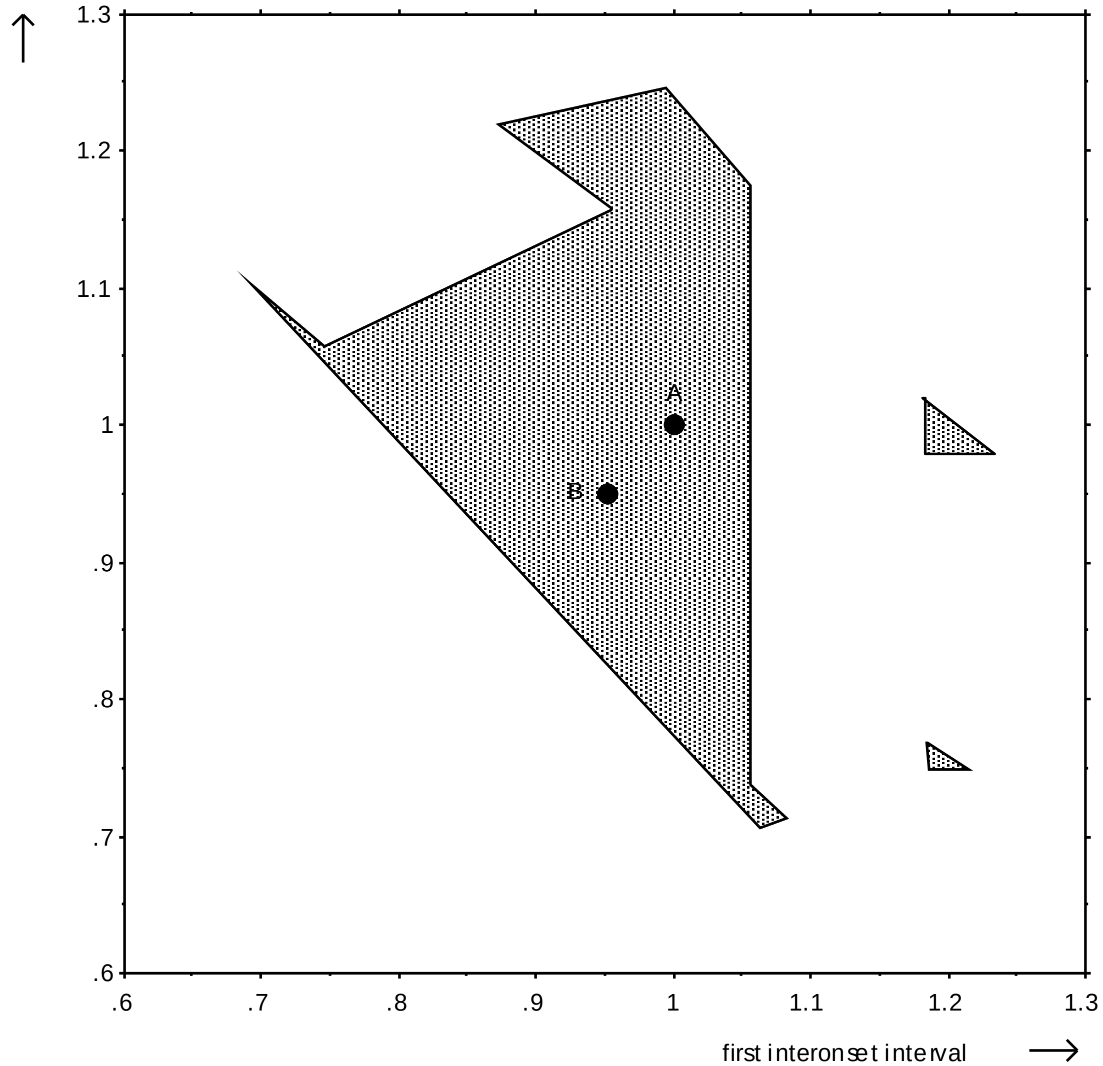

Figure 3. *Temporal patterns interpreted as triplets.*

In Figure 3 a map of all possible subdivisions of a fixed beat length into three notes is given with the region that is identified by the parser as a triplet. The size of that region depends on the tollerance (which was taken as one tenth of the beat period). The initial pulse at that level was taken to be two. The idealized metronomical triplet is located on the map at a point marked A. Given the performance of the parser it is not surprising that the actual form of the region is biased towards the pattern (short, short, long) found by Vos and Handel (1987) in their study of systematic deviations from the norm in their group of subjects who could play triplets well. This typical triplet pattern is located at point B. The same reasoning may be used to relate the behavior of the parser to empirical findings at higher metrical levels where elongation towards the end of the unit seems to be the rule (Todd, 1985 and Clarke, 1987). This knowledge about common performance practice implicitly incorporated in the model, which may explain its success, was not identified in the original article.

The unconnected small regions that also signify patterns parsed as triplets are a by-product of the interaction of the tempo tracking and the mechanism for meter chance. While the former allows for a large area of triplet parsing extending to the right, the latter decides for a duple meter in most parts of that area, but fails to do so for the two small islands. A large tollerance will enlarge the islands, finally linking them up with the main region. But in general it can be said that the equivalence classes induced by the parser, the set of temporal patterns that will be interpreted as performances of the same rhythm, do not form one connected region. I could not find indications about the plausibility of this result in the literature about human rhythm perception, but it will greatly complicate empirical verification of the model.

<u>change of meter</u>

The decision when to change meter is taken, among others, on the basis of an syncope occuring in the last subdivision. This sometimes seems too restricted, as syncopes in the other subdivisions might also contribute evidence for a change of meter. A more sophisticated model might adapt the reluctance to change the meter, to the metrical level in consideration, making the higher levels which resemble time signature less prone to changes than the lower levels. In the new program it is easy to experiment with this and other variants but in general the question seems very difficult to solve.

conclusion

One can ponder what the truth is in the following quotation about the procedures in the musical parser:

> *"Such procedures are, unfortunately, much more difficult to specify precisely in English than in a suitably designed programming language: but this fact only underlines the value of casting perceptual theories in computational form"* (Longuet-Higgins, 1987 p. 109)

But if a programming language allows the programmer to express such difficult constructs in a program, will it then be possible to see the ramifications of the theory? Or has theory degenerated into a black box mechanism that can only be used, instead of understood. I tend to attribute more value to the adagio:

> *"If you can't write it down in English, you can't code it"* (Peter Halpern in Bentley, 1988, p. 58)

But because moulding a theory into an implementation greatly helps in understanding and describing the theory in plain English, a computational approach in which the process of developing theory and implementing go hand in hand, is still most attractive to most AI researchers. That the resulting program often contains vestigial remains of earlier versions (Longuet-Higgins, personal communication) just calls for one more round of cleaning up and rewriting, as I hope to have shown in this article. The rewrite, at first sight a scholarly exercise, soon became a major undertaking because of the tangled flow of control and data in the program. But finally the program was made much more open for experimentation, verification or falsification and possibly extension. It is now easier to maintain and immerse in systematic testing, the more so since the algorithm was implemented in POCO (Honing 1990), an environment for research in expressive timing. In the process of rewriting, semantic invariant program transformations turned out to be very helpful as a methodology for reverse engineering as was the availability of a test suite to automate some test runs after each change.

I think that computational psychology can be a fruitful approach to the study of music, complementing musicology, experimental psychology and other disciplines. But to play this role well, researchers must force themselves to state their algorithmic contributions in the form of clean micro-programs and clarify which parts of the program are considered to model cognitive processes and which parts are implementation detail or technical tricks.

acknowledgements

Christopher Longuet-Higgins is the first to be thanked here. His work is a continuous source of inspiration, and his encouragement was a great help in this research. With help from Henkjan Honing the first attack of the POP-2 code was made, and some of his good ideas were used in this article. This research was partly supported by an ESRC grant under number A413254004.

references

Abelson, H., G.J. Sussman. 1985 Structure and Interpretation of Computer Programs. Cambridge, MA: The MIT Press.

Barett R., A. Ramsay and A.Sloman. 1985 "POP-11, A Practical Language for Artificial Intelligence" Chichester: Ellis Horwood Ltd.

Burstall, R.M. and J.S. Colins and R.J. Poppelstone. 1968 "POP-2 papers" Edinburgh: University Press.

Bentley J. 1986 "Programming Pearls" Reading, MA: Addison-Wesley.

Bentley J. 1988 "More Programming Pearls, Confessions of a Coder" Reading, MA: Addison-Wesley.

Campbell, J.A. 1990 " Three novelties of AI: theories, programs and rational reconstruction" In: The foundations of artificial intelligence. A source book, edited by D. Partridge and Y. Wilks. Cambridge: Cambridge University Press.

Clarke, E.F. 1987 "Levels of Structure in the Organisation of Musical Time" in: Music and Psychology, a Mutual Regard. S. McAdams(Ed.) Contemporary Music Review 2(1).

Desain, P. and H. Honing , (1991). "The Quantization Problem: Traditional and Connectionist Approaches. " in Balaban,M., K. Ebcioglu & O. Laske, eds Musical Intelligence. Menlo Park: The AAAI Press.(forthcomming)

Desain, P. and H. Honing , 1989. "Quantization of Musical Time: A Connectionist Approach. " Computer Music Journal 13(4) reprinted in..Todd, P.M. & D.G. Loy (Eds.) (1991) Music and Connectionism, Cambridge, Mass.: MIT Press. (forthcoming).

Desain P. 1990. "Lisp as a Second Language, Functional Use" Perspectives of New Music. 27(1).

Desain, P. 1991 "A Connectionist and a Traditional AI Quantizer, Symbolic versus Sub-symbolic Models of Rhythm Perception" In I. Cross (Ed.), Proceedings of the 1990 Music and the Cognitive Sciences Conference, Contemporary Music Review. London: Harwood Press.

Dijkstra, E.W. 1968 "GOTO statement Considered Harmful" Comm. ACM 11(3)

Honing, H. 1990 "POCO: An Environment for Analysing, Modifying, and Generating Expression in Music" In *Proceedings of the 1990 International Computer Music Conference.* San Francisco: Computer Music Association.

Longuet-Higgins, H.C., 1976. "The Perception of Melodies" Nature 263: 646-653 and in Longuet-Higgins, 1987.

Longuet-Higgins, H.C., 1979. "The Perception of Music" Proc. R. Soc. Lond. B 205: 307-322 and in Longuet-Higgins, 1987.

Longuet-Higgins, H.C., 1983 "All in theory, the analysis of music" Nature 304(7) p 93.

Longuet-Higgins, H.C., 1987. Mental Processes. Cambridge, Mass.:MIT Press.

Shank R.C. and C.K.Riesbeck. 1981 "Inside Computer Understanding" Hillsdale, New Jersey: Lawrence Erlbaum.

Steele, G.L.Jr 1984. Common Lisp: the Language. Burlington, MA:
Digital Press.

Todd, N.P., 1985. A Model of Expressive Timing in Tonal Music. *Music Perception* 3(1):33-58.

Vos, P. and S. Handel 1987. "Playing triplets: Facts and Preferences" in A. Gabrielson (Ed.) Action and perception in Rhythm and Music. Royal Swedish Academy of Music. 55.

Appendix 1: relevant parts of original POP-2 code

```
recordclass note pitch onset offset ..extra fields declared here.
function sift notefile=>notefile;
  maplist(notefile, lambda x;
  if x.tl.tl.hd-x.tl.hd<5 then else x.close
  end)->notefile;
end;

function takein notefile=>nlist;
  maplist (notefile, lambda x;
       consnote (applist(x, identfn), undef, undef, undef)
  end)->nlist;
end;

functions res, int, modulate, hark, simplify, intervals, tuneup and vars
flag,k,l,m,n,place declared here

vars start beat position number group last metre nlist sequence;

function startup;
  nil->sequence; nlist.hd.onset->start;
  nlist.tl,hd.onset-start->beat;
  nlist.hd.pitch->position;
  nil->group; nil->last; 0->number;

    loopif nlist.hd.pitch=position then
       nlist.tl->nlist; number+1->number
    close;
end;

vars tol metre; 13->tol; nil->metre;

function singlet->stop->fig;
    vars period mark;
    if group.null.not then
        if group.hd.offset<stop-period/2 then "stc"
        elseif group.hd.offset<stop-tol then "ten"
        else "leg"
        close->mark;

        group.rev->last; nil->group; mark::last;
    else
        [%"tac",applist(last,lambda x;
        if x.offset>start+tol then x
        close end)%]
    close->fig;
    if nlist.null or nlist.hd.onset>stop+tol then 0
    else nlist.hd.onset
    close->stop;
end;

function rhythm start period->stop->fig; vars stop;
  start+period->stop;
  if nlist.null.not and nlist.hd.onset<start+tol
  then nlist.hd::nil->group; nlist.tl->nlist;
  else goto label
  close;
  loopif nlist.null.not and nlist.hd.onset<stop+tol
      and nlist.hd.onset<group.hd.onset+tol
```

```
  then nlist.hd::group->group; nlist.tl->nlist;
  close;
  if group.hd.onset>stop-tol
  then group.hd::nlist->nlist; group.tl->group
  close;
label;
  if nlist.null or nlist.hd.onset>stop-tol
  then .singlet
  else .tempo
  close->stop->fig;
end;

function tempo->stop->figure;
  vars new old again pulse time count fig syncop;

  [%nlist,last,group%]->old; 0->again;
loop:
  if metre.null then 2::nil->metre
  close;

  metre.hd->pulse; metre.tl->metre;
  nil->figure;  period->time;
  0->count; start->stop;
  loopif count<pulse
  then
      count+1->count;
      rhythm(stop, time/pulse)->stop->fig;
      fig::figure->figure;
      if stop=0 then start+count*time/pulse->stop; true
      else stop-start+(pulse-count)*time/pulse->time; false
      close->syncop;
  close;
  again+1->again;

  if not (syncop or stop>start+period+tol or stop<start+period-tol)
  then figure.rev->figure; pulse::metre->metre
  exit;
  if again=1 then
      [%nlist,last,group,figure.rev,stop,pulse::metre%]->new;
      old.destlist->group->last->nlist;
      (5-pulse)::nil->metre; goto loop;
  else
      new.destlist->metre->stop->figure->group->last->nlist;
  close;
end

function tapout nlist->sequence;
  vars start beat tol group last stop figure;
  loopif nlist.null.not
  then
      rhythm (start, beat)->stop->figure;
      figure::sequence->sequence;

      if stop=0 then start+beat
      else (stop-start+beat)/2->beat; stop
      close->start;
  close;
  nil->metre;
  sequence.rev->sequence;
end;
```

```
vars max,min,symbols,symbol declared and initialzed here
function name declared here

function describe fig; vars word;
  fig.hd->word; fig.tl->fig;
  if fig.null then [rest]
  elseif word="tac" then
      "tied"::maplist(fig,index<>symbol)
  elseif word="leg" then maplist(fig, name)
  else [%applist(fig,name),word%]
  close;
end;

function reveal figure;
  if figure.hd.isword
  then figure.describe
  else maplist(figure,reveal)
  close;
end;

function typeout seq; vars count;
  0->count; 1.nl;
  applist(seq,lambda x;
     if count = number then 1->count; 2.nl
     else count+1->count
     close; x.reveal .pr
  end); 2.nl;
end

function notate notefile;
  notefile.takein->nlist;
  .startup;
  nlist.tapout->sequence;
  nlist.tuneup;
  sequence.typeout;
end;
```

Appendix 2

```
;;; Longuet-Higgins Musical Parser,
;;; Micro-version 2, in Common Lisp (uses loop macro), Peter Desain, 1991.
;****************************************************************************
; top level

(defvar *tollerance*)

(defun notate (note-list &key
                         (metre '(2))
                         (tollerance 10)
                         (start (onset (first note-list)))
                         (beat (- (onset (second note-list))
                                  (onset (first note-list))))
                         (speed 0.5))
  (setf *tollerance* tollerance)
  (loop while note-list
        with figure
        with group = nil
        do (multiple-value-setq
            (start figure group metre note-list beat)
            (rhythm start beat group metre note-list (+ start beat) speed))
        collect figure into figures
        finally (return (articulation figures))))

;****************************************************************************
; main parsing routines

(defun rhythm (start period group metre note-list aim speed)
  (let ((stop (+ start period)))
    (multiple-value-bind (group note-list)
                         (collect-group group note-list start stop)
      (if (or (null note-list)(not (onset-before (first note-list) stop '-)))
        (singlet start period group metre note-list aim speed)
        (tempo start period group metre note-list aim speed)))))

(defun singlet (start period group metre note-list aim speed)
  (let* ((stop (+ start period))
         (syncope (or (null note-list)
                      (not (onset-before (first note-list) stop '+))))
         (end (if syncope aim (onset (first note-list)))))
    (values end (list start stop period group) nil metre note-list
            (+ period (* speed (- end aim))) syncope)))

(defun tempo (start period group metre note-list aim speed)
  (apply #'values
   (rest (generate-and-test
                      #'trial
                      (make-test (+ start period))
                      #'alternative
                      metre start period group note-list aim speed))))

(defun make-test (aim)
  #'(lambda (syncope stop &rest ignore)
      (and (not syncope)
           (< (abs (- stop aim)) *tollerance*))))

(defun alternative (metre &rest arguments)
  (cons (alternative-metre metre) arguments))
```

```
;*****************************************************************************
; control structure for change of metre

(defun generate-and-test (generate test alternative &rest states)
  (let ((result1 (apply generate states)))
    (if (apply test result1)
      result1
      (let ((result2 (apply generate (apply alternative states))))
        (if (apply test result2)
          result2
          result1)))))

;*****************************************************************************
; subdivide a period

(defun trial (metre start period group note-list aim speed)
  (loop
    with pulse = (pop metre)
    with sub-start = start
    with sub-period = (/ period (float pulse))
    with syncope
    for count from 1 to pulse do
    (multiple-value-setq
     (sub-start fig group metre note-list sub-period syncope)
     (rhythm sub-start sub-period group (extent-metre metre) note-list
             (+ start (* count sub-period)) (/ (float pulse))))
    collect fig into figure
    finally (return (list syncope sub-start figure group (cons pulse metre)
                          note-list (+ period (* speed (- sub-start aim)))))))

;*****************************************************************************
; metre calculus

(defun alternative-metre (metre)
  (case (first metre)
    (2 '(3))
    (3 '(2))))

(defun extent-metre (metre)
  (or metre '(2)))

;*****************************************************************************
; collect group of synchronous notes

(defun collect-group (group note-list start stop)
  (if (and note-list (onset-before (first note-list) start '+))
    (collect-new-group (list (first note-list)) (rest note-list) stop)
    (values group note-list)))

(defun collect-new-group (group note-list stop)
  (if (and (collect-group-test (first note-list)(first group) stop)
           (or (collect-group-test (second note-list)(first note-list) stop)
               (onset-before (first note-list) stop '-)))
    (collect-new-group (cons (first note-list) group) (rest note-list) stop)
    (values (reverse group) note-list)))

(defun collect-group-test (note1 note2 stop)
  (and note1
       (onset-before note1 stop '+)
       (onset-before note1 (onset note2) '+)))
```

```lisp
;*****************************************************************************
; articulation analysis

(defun articulation (l &optional last)
  (cond ((null l) (values nil last))
        ((listp (first l))
         (multiple-value-bind (result1 last1)
                              (articulation (first l) last)
           (multiple-value-bind (result2 last2)
                                (articulation (rest l) last1)
             (values (cons result1 result2) last2))))
        (t (apply #'articulate-figure last l))))

(defun articulate-figure (last start stop period group)
  (let* ((new-last (or group (remove-if #'(lambda(note)
                                             (offset-before note start '+))
                                         last)))
         (pitches (mapcar #'pitch new-last)))
    (values (figure-describe group stop period pitches) new-last)))

(defun figure-describe (group stop period pitches)
  (if (null group)
    (if pitches (cons 'tied pitches) '(rest))
    (append pitches (articulation-mark (first (last group)) stop period))))

(defun articulation-mark (note stop period)
  (cond ((offset-before note (- stop (/ period 2.0)))
         '(stc))
        ((offset-before note stop '-)
         '(ten))
        (t nil)))

(defun snoc (l x)(nconc l (list x)))

;*****************************************************************************
; help functions

(defun onset-before (note time &optional (margin 0))
  (< (onset note)  (+ time (case margin
                             (+ *tollerance*)
                             (- (- *tollerance*))
                             (otherwise 0)))))

(defun offset-before (note time &optional (margin 0))
  (< (offset note) (+ time (case margin
                             (+ *tollerance*)
                             (- (- *tollerance*))
                             (otherwise 0)))))

;*****************************************************************************
; data abstraction for notes

(defstruct (note (:constructor note (pitch onset offset))
                 (:conc-name nil))
  pitch onset offset)
```

```
;***************************************************************************
; example of the use of the program
#|
; defining a note list
(defvar *cliche* (list (note 'start 154 227)
                       (note 'c 285 294)
                       (note 'g 322 327)
                       (note 'g 336 341)
                       (note 'as 349 383)
                       (note 'g 384 407)
                       (note 'b 445 453)
;                       (note 'c 484 527)))

;calling the program:
(notate *cliche* :tollerance 10)

; will produce the following results:
((START TEN)
 (((C STC)
   ((G STC)
    (G STC)))
  ((AS) (G TEN)))
 (((REST) (B STC))
  (C TEN)))

|#
```

Appendix 3: Test Data

Note	Onset	Offset
START	24	114
START	148	238
C	274	399
G	400	554
BB	551	587
AB	586	671
EB	669	711
AB	707	794
D	795	831
G	829	860
C	863	895
F	895	989
G	987	1021
F	1020	1145
EB	1140	1242
D	1268	1282
C	1289	1298
BB	1308	1320
F	1332	1452
D	1450	1495
BB	1508	1517
A	1528	1536
AB	1546	1556
EB	1570	1696
C	1692	1734
AB	1752	1762
G	1774	1782
FS	1792	1808
D	1815	1930
F	1928	1934
EB	1932	2062
D	2059	2188
DB	2183	2446
C	2491	2628

The TRIS example: a fragment of the cor anglais solo in the Prelude to Act III of Wagner's *Tristan und Isolde*.

[insert musical fragment about here]

Note	Onset	Offset
START	148	190
G	280	287
F	302	309
EB	322	329
BB	347	466
G	474	518
EB	538	548
D	559	566
CS	578	586
A	605	648
FS	646	657
D	669	678
CS	687	696
C	707	714
AB	729	760
F	769	777
DB	791	801
C	811	820
B	830	839
G	856	987
EB	986	1027
C	1049	1054
B	1068	1075
BB	1087	1096
F	1111	1153
D	1152	1157
BB	1174	1183
A	1194	1202
AB	1211	1220
EB	1232	1270
C	1272	1279
AB	1295	1304
G	1316	1325
FS	1336	1348
D	1360	1619

The STAN example: a fragment of the cor anglais solo in the Prelude to Act III of Wagner's *Tristan und Isolde*.

[insert musical fragment about here]

Note	Onset	Offset
START	154	227
C	285	294
G	322	327
G	336	341
AS	349	383
G	384	407
B	445	453
C	484	527

The CLICHE example

[insert musical fragment about here]